Java Packages

Introduction to Java Packages

- Java contains many predefined classes that programmers can re-use rather than "reinventing the wheel". These classes are grouped into categories of related classes called packages.

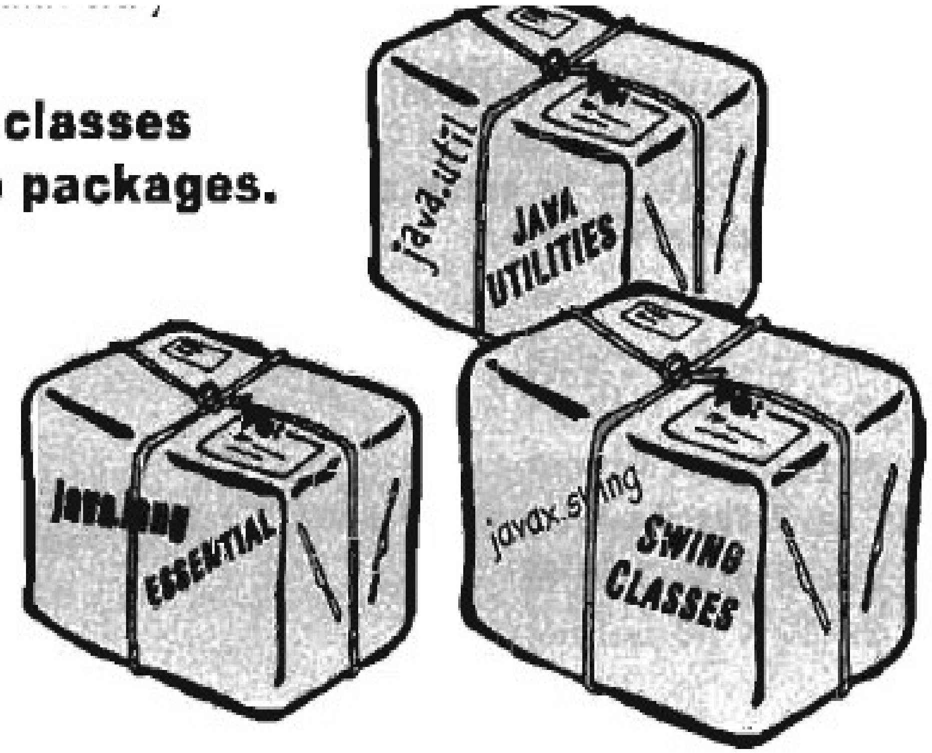
  Together, we refer to these packages as the Java Application Programming Interface (Java API), or the Java class library.

- The Java Package name consists of words separated by periods. The first part of the name represents the organization which created the package. The remaining words of the Java Package name reflect the contents of the package.

- for example, the Java-supplied package "java.lang". The first part of that package name "java“ represents the organization that developed the package (Sun's **Java group**). The second part of the package name "lang" is basic language functionality and fundamental types.

Introduction to Java Packages

In the Java API, classes are grouped into packages.



Java Packages

A package represents a directory that contains related group of classes and interfaces

A package is a container for related classes and interfaces, providing access protection and namespace management. Thus, packages enable grouping of functionally related classes and interfaces

In simple words, packages is the way we organize files into different directories according to their functionality, usability as well as category they belong to.

Java has two different types of packages:

1.Built-in packages

These are packages which are already developed by the Java development team.

Details of their classes and interfaces can be found in the Java Documentation

2. User defined packages

Examples of Java built-in Packages

Package	Description
`java.applet`	***The Java Applet Package.*** **This package contains the `Applet` class and several interfaces that enable the creation of applets, interaction of applets with the browser and playing audio clips. In Java 2, class `javax.swing.JApplet` is used to define an applet that uses the *Swing GUI components*.**
`java.awt`	***The Java Abstract Windowing Toolkit Package.*** **This package contains the classes and interfaces required to create and manipulate graphical user interfaces in Java 1.0 and 1.1. In Java 2, these classes can still be used, but the *Swing GUI components* of the `javax.swing` packages are often used instead.**
`java.awt.event`	***The Java Abstract Windowing Toolkit Event Package.*** **This package contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages.**
`java.io`	***The Java Input/Output Package.*** **This package contains classes that enable programs to input and output data**
`java.lang`	***The Java Language Package.*** **This package contains classes and interfaces required by many Java programs (many are discussed throughout this text) and is automatically imported by the compiler into all programs.**

Examples of Java built in Packages

Package	Description
`java.net`	*The Java Networking Package.* This package contains classes that enable programs to communicate via networks
`java.text`	*The Java Text Package.* This package contains classes and interfaces that enable a Java program to manipulate numbers, dates, characters and strings. It provides many of Java's internationalization capabilities i.e., features that enable a program to be customized to a specific locale (e.g., an applet may display strings in different languages, based on the user's country).
`java.util`	*The Java Utilities Package.* This package contains utility classes and interfaces, such as: date and time manipulations, random-number processing capabilities (`Random`), storing and processing large amounts of data, breaking strings into smaller pieces called *tokens* (`StringTokenizer`) and other capabilities
`javax.swing`	*The Java Swing GUI Components Package.* This package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs.
`javax.swing.event`	*The Java Swing Event Package.* This package contains classes and interfaces that enable event handling for GUI components in the `javax.swing` package.

Using Packages

Note: To gain access to any of these packages, except for java.lang package which is always pre-imported, the following syntax is used:

```
import java.package.*;
```

- The asterix (*), indicates that all classes in that particular package are made available.

For example: `import java.util.*;`

All classes in the util package can be accessed.

```
import java.util.Random;
import java.util.Date;
```

Only the two classes (Random and Date) can be accessed.

Interactive Java Programs (Console I/O)

Java API can facilitate to design programs that can read from a user interactively during execution.

```
// The following program calculates the sum of two integers.
      import java.util.Scanner; // program uses class Scanner
     public class Product {
     public static void main( String[]  args) {
//create scanner object to obtain input from command window
Scanner scn = new Scanner (System.in);
     int x; //first number input by user
     int y; // second number input by user
      int sum; // sum of two numbers
 // prompt for input
System.out.println ("Enter first integer:" );
x = scn.nextInt( ); // read first integer
```

Interactive Java Programs (Console I/O)

```
// prompt for input
    System.out.println("Enter second integer:" );
    y = scn.nextInt( ); //read second integer

     sum = x + y; // calculate sum of numbers

    System.out.println("The sum of the two numbers is: " +
    sum );

            } //end main method

     }// end class Product
```

User defined Packages

These are created by users who also decide which classes and interfaces need to be part of such packages

The general syntax for creating packages is:

```
package pakageName;
```

Packages are mirrored through a directory structure, To create a package, first we have to create a directory structure that matches the package hierarchy

Package structure should also match the directory structure

To make a class belong to a particular package, include the package statement as the first statement of source file

User defined Packages-Example

```
package pack;
public class Addition
{
    private double d1,d2;
    public Addition(double a,double b)
{
    d1=a;
    d2=b;
}
    public void sum()
{
    System.out.println(“The sum is:”+(d1+d2));
}
}
```

User defined Packages-Example

- Using the import statement, the class Use could be rewritten as follows:

```
import pack.Addition;
class Use
{
public static void main(String arg[])
{
Addition obj=new Addition(25, 54);
obj.sum();
}
}
```

User Defined Packages-Example

To use the class contained in the package

```
class Use
{
public static void main(String arg[])
{
pack.Addition obj=new pack.Addition();
obj.sum();
}
}
```

The above program will run without the use of an import statement to include the contents of the package pack in the program which has the class use.

User Defined Packages-Example

Another class namely Subtraction can be added to our package pack as follows:

```
package pack;
public class Subtraction
{
    public Subtraction(double a, double b)
{
public static double sub( )
{
return (a-b);
} }  }
```

After compiling this class using the same procedure as above, subtraction class will be added into the directory pack and it can be used effectively in a program by importing it.

Naming Convention

- If you do not use a package statement, your type ends up in an unnamed package.
- Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process.
- Otherwise, classes and interfaces belong in named packages.
- Same names in different packages works well unless two independent programmers use the same name for their packages. What prevents this problem? = Name Convention.

Naming Conventions

Package names are written in all lower case to avoid conflict with the names of classes or interfaces.

Companies use their reversed Internet domain name to begin their package names—for example, `com.example.mypackage` for a package named `mypackage` created by a programmer at `example.com`.

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, `com.example.region.mypackage`).

Packages in the Java language itself begin with `java.` or `javax.`

Summary of Creating and Using Packages

To create a package for a type, put a `package` statement as the first statement in the source file that contains the type (class, interface, enumeration, or annotation type).

To use a public type that's in a different package, you have three choices: (1) use the fully qualified name of the type, (2) import the type, or (3) import the entire package of which the type is a member.

The path names for a package's source and class files mirror the name of the package.

Question

1. Assume you have written some classes. Belatedly, you decide they should be split into three packages, as listed in the following table. Furthermore, assume the classes are currently in the default package (they have no package statements).

Package Name	Class Name
mygame.server	Server
mygame.shared	Utilities
mygame.client	Client

a. Which line of code will you need to add to each source file to put each class in the right package?
b. To adhere to the directory structure, you will need to create some subdirectories in the development directory and put source files in the correct subdirectories. What subdirectories must you create? Which subdirectory does each source file go in?
c. Do you think you'll need to make any other changes to the source files to make them compile correctly? If so, what are the changes?

Answers

Answer 1a: The first line of each file must specify the package:

In `Client.java` add:

```
package mygame.client;
```

In `Server.java` add:

```
package mygame.server;:
```

In `Utilities.java` add:

```
package mygame.shared;
```

Answer 1b: Within the `mygame` directory, you need to create three subdirectories: `client`, `server`, and `shared`.

In `mygame/client/` place:
`Client.java`

In `mygame/server/` place:
`Server.java`

In `mygame/shared/` place:
`Utilities.java`

Answer 1c: Yes, you need to add import statements. `Client.java` and `Server.java` need to import the `Utilities` class, which they can do in one of two ways:

```
import mygame.shared.*;
       --or--
import mygame.shared.Utilities;
```

Also, `Server.java` needs to import the `Client` class:

```
import mygame.client.Client;
```

Exceptions Handling

Exceptions Handling

Mistakes happens! The file isn't there, The server is down. No matter how good a programmer you are, you can't control everything. Things can go wrong, sometimes very wrong!.

When you write a risky method, you need codes to handle the bad things that might happen.

But how do you know if a method is risky? And where do you put the code to handle the risky situation?

Problems happening during runtime are mostly flaws in a code, such as bugs, that should be fixed at development time.

The exceptions in this case are for code with no guarantee that they will work at runtime. For example, codes that expects the file to be in the right directory, the server to be running etc etc…

Exceptions Handling

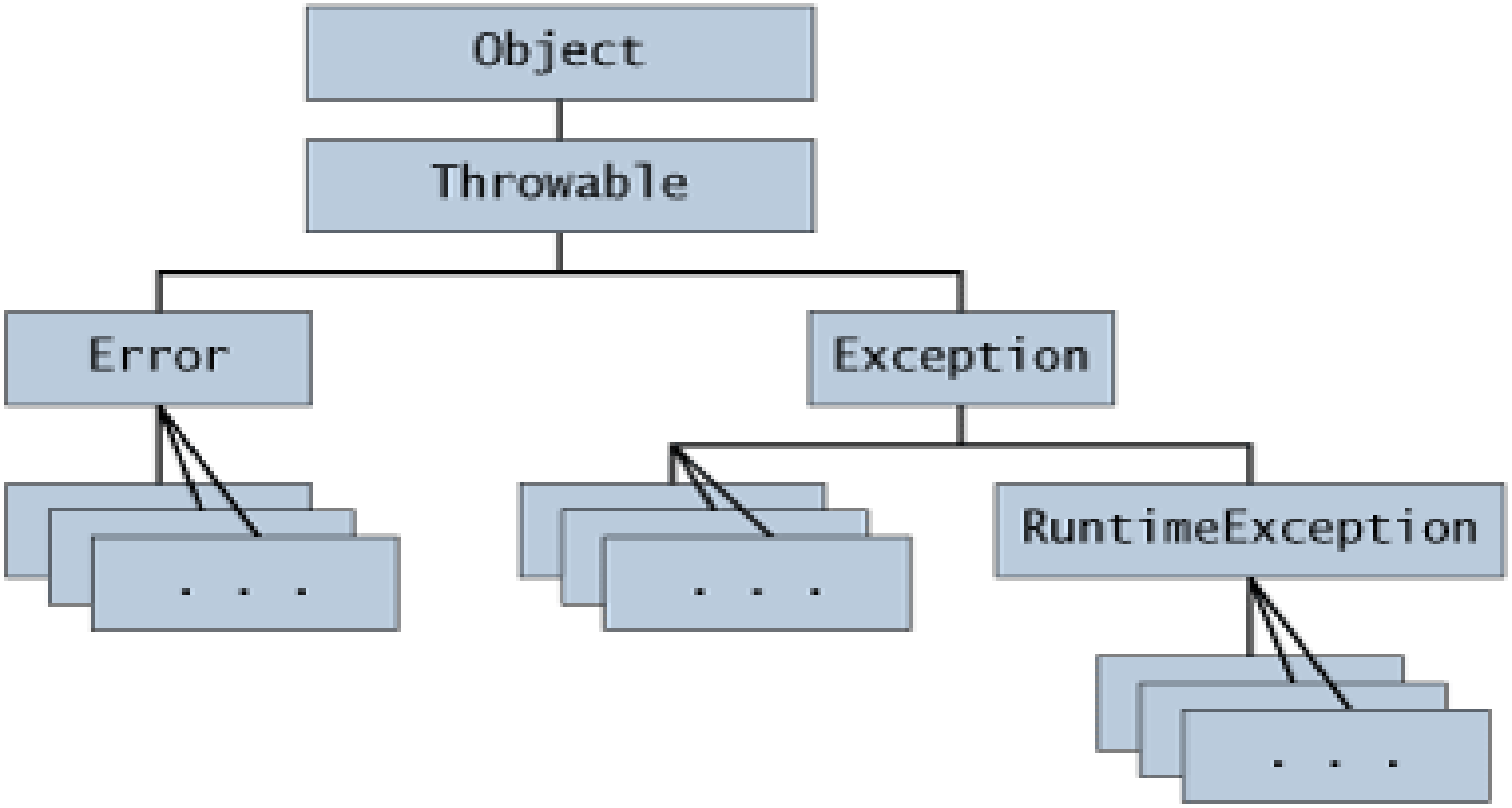
- A java exception is an object that describe an abnormal condition that has occurred in a piece of code (or in a certain method). It occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

- When an exceptional condition arises, an object representing such exception is created and thrown in the method that caused the problem. The method can choose to handle the exception itself or pass on, either way at some point the exception is caught and processed.

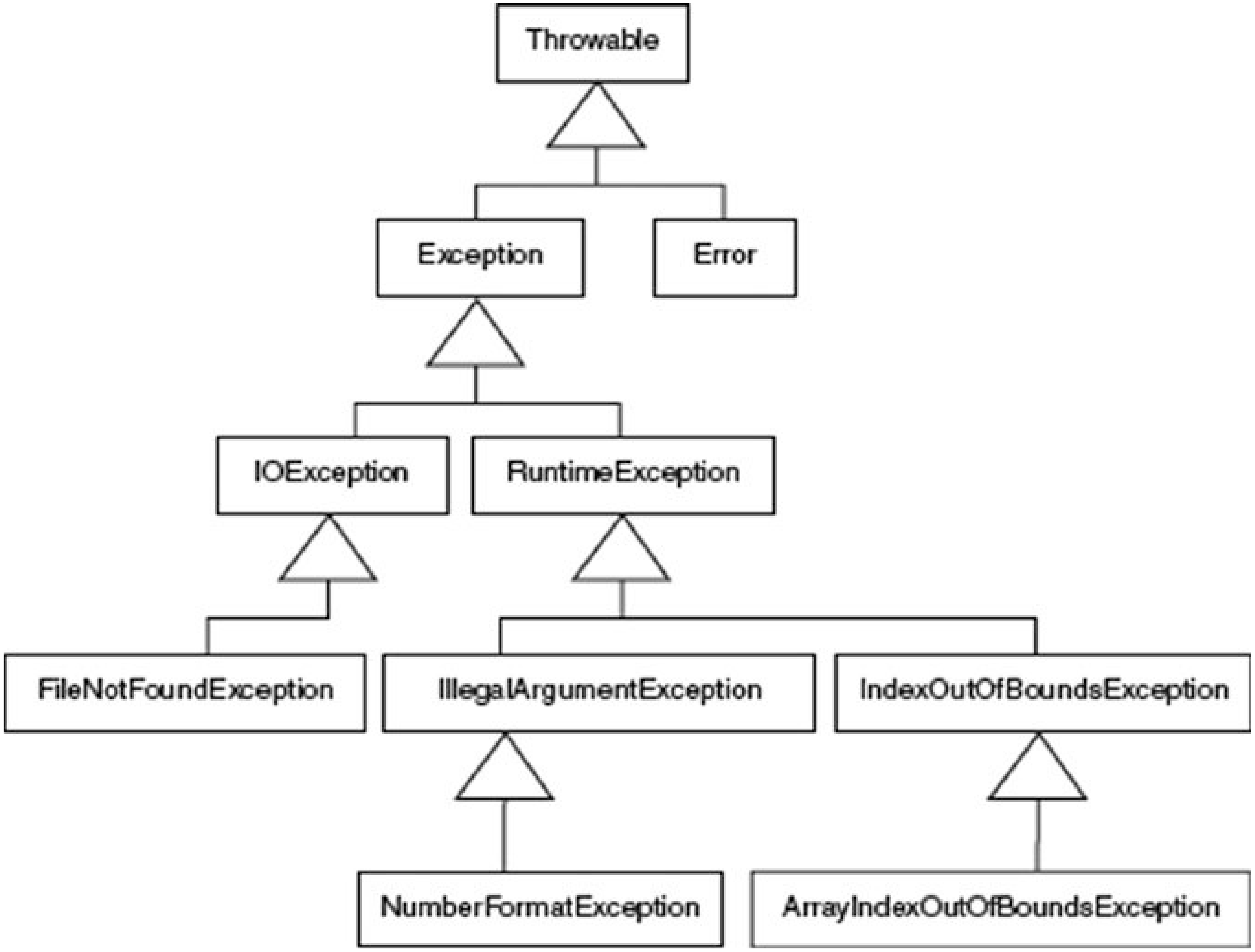
- Errors are object created by the java run-time system to indicate problems having to do with the run-time environment itself. Error objects are created to respond to catastrophic failures that cannot be handled by a user program.

Exceptions Handling

- Both errors and exceptions are the subclass of the built-in class Throwable.



Exceptions Handling



Exceptions Handling

- The RuntimeException class deals with errors that arise from the logic of a program. Since nearly every Java instruction could result in a RuntimeException error, the Java compiler does not flag such instructions as potentially error-prone.
- Consequently these types of errors are known as **unchecked exceptions**. It is left to the programmer to ensure that code is written in such a way as to avoid such exceptions; for example, checking an array index before looking up a value in an array with that index. Should such an exception arise, it will be due to a program error and will not become apparent until runtime.

Checked & Unchecked Exceptions

- The Java compiler usually flag up those instructions that may generate all other types of IOException , since the programmer has no means of avoiding such situations to happen. For example, an instruction to read from a file may cause an exception because the file is corrupt or the file is not there. No amount of program code can prevent this file from being corrupt.

- The compiler will flag such an instruction as potentially error-prone, it will also specify the exact exception that could be thrown. These are known as **checked exceptions.**
- Programmers have to include code to inform the compiler of how they will deal with checked exceptions generated by a particular instruction, before the compiler will allow the program to compile successfully.

Exceptions Handling

- The java exception handling is managed via five keywords: **try, catch, throw, throws** and **finally**.
  - Program statements to monitor for exceptions are contained in the **try** block. If an exception occur within a try block , it is thrown.
  - A user program can catch this exception by using the **catch** block and handle it.
  - The keyword **throw** is used to manually throw an exception.
  - Any exception that is thrown out of a method must be specified by a **throws** clause.
  - **finally** block contain all codes that must be executed before the try block ends.

Exceptions Handling: try-catch blocks

- The **try-catch** block structure

  try {

  //statements that may throw an exception

  }

  catch (*ExceptionType exceptionReference*) {

  //statements to process an exception

  }

- A **try** block can be followed by any number of **catch** blocks

Exceptions Handling: Uncaught Exceptions.

- The following program illustrates what happens when a program is written without any exception handling mechanism. The following program causes a division by zero.

```
class Exc0 {
public static void main (String [] args) {
        int d = 0;
        int a = 30/d;
        }
}
```

- When java run-time system detects the attempt to divide by zero, it constructs and throw an exception object.
- This causes the execution of Exc0 to stop, because once the exception has been thrown it must be caught by an exception handler.

Exceptions Handling: Uncaught Exceptions

- But the program hasn’t supplied any exception handler of its own, instead the exception will be caught by the default handler of the java run-time system.
- The default handler terminates the program and display a string describing the exception as shown below:

```
java.lang.ArithmeticException: / by zero
                at Exc0.main(Exc0.java:4)
```

Exceptions Handling: Using try and catch

The following program include **try** and **catch** which handles the **ArithmeticException** generated by divide-by-zero exception.

```
class Exc1 {
    public static void main (String [] args) {
                int d, a;
 try { // monitor a block of code.
                 d = 0;
                 a = 42 / d;
                  System.out.println (“The answer is” + a);
                } catch (ArithmeticException e) { //catch the division -by-
    zero exception
                   System.out.println (“Division by zero.”);
                }
    }
}
Output: Division by zero.
```

Exceptions Handling: Multiple catch clauses.

- Note: In case of multiple catch clauses, catch statement that uses a superclass will catch exceptions of all that class plus any of its subclasses. Thus, in this case a subclass catch will never be reached.

```
class SuperSubCatch {
    public static void main (String [] args) {
        try { // monitor a block of code.
                int a  = 0;
                int c = 42 / a;
    } catch (Exception e) {
                  System.out.println (“Catching generic exceptions.”);
                }
    catch (ArithmeticException ae) {
                  System.out.println (“Catching arithmetic exceptions.”);
                }
    }
}
```

Exceptions Handling: Multiple catch clauses

- The previous program with multiple catch clause will never compile, an error message will be displayed stating that the second catch statement is unreachable because the exception has already been caught.

- To fix the problem, reverse the order of the catch statements.

```
class SuperSubCatch {
            public static void main (String [] args) {
               try { // monitor a block of code.
                        int a  = 0;
                        int c = 42 / a;

            } catch (ArithmeticException ae) {
            System.out.println (“Catching arithmetic exceptions.”);
                        }
            catch (Exception e) {
            System.out.println (“Catching generic exceptions.”);
                        }
            }
}
```

Exceptions Handling

```
import java.util.Scanner;

public class Error_handling {
static int x;
static String y;

public static void main(String[] args) {
Scanner input = new Scanner (System.in);

System.out.println("Please enter an integer value");
                        x = input.nextInt();

                        System.out.println("You entered:" + x);

                        try {
                        x = input.nextInt();
                        }
catch (Exception e)    {
                        System.out.println("Wrong input"+e);
                         System.out.println("Wrong input, an integer is expected");


    }
                        System.out.println("You entered:" + x);
     }
}
```

Exceptions Handling

```
import java.util.Scanner;

public class Error_handling2{
 static int x;

public static int getAge()    {
      Scanner input = new Scanner (System.in);
      System.out.println("Please enter your age");
      try {
      x = input.nextInt();
      }catch (Exception e)
       {System.out.println("Sorry, wrong input, expecting an integer number");
            getAge();                    }
                 return x;
      }

    public static void main(String[] args)
   {
      getAge();
      System.out.printf("You are %d years old", x);
   }
}
```

Using throw and throws

- So far we have been catching exceptions that are thrown by the java run-time system. However a program can throw an exception explicitly using a **throw** statement.
  The general form of a **throw** statement is:

```
throw ThrowableInstance;
```

- *ThrowableInstance* must be an object or a subclass of type **Throwable.** Simple types such as **int** or **char**, as well as **non-throwable** classes cannot be used.
- There are two ways to obtain **Throwable** instances: using a parameter into a **catch** clause or creating one with the **new** operator.

Using throw and throws

- If a method is capable of causing an exception that it does not handle, it has to specify this behavior so that callers of such a method can guard themselves against this exception.
- This is done by including a **throws** clause in the method's declaration. The general form of a method declaration that include a throws clause:

```
type method-name(parameter-list) throws exceptions-list
{
        //body of method
}
```

- *Exception-list* is a comma separated list of the exceptions that a method can throw

Using throw and throws

- A sample program that uses throws clause.

```
class ThrowsDemo {
        static void methOne () throws IllegalAccessException {
         System.out.println (“ Inside method one”);
        throw new IllegalAccessException(“demo”);
        }
        Public static void main (String [] args) {
          try {
              methOne();
        } catch (IllegalAccessException e) {
         System.out.println (“Caught: “ + e); }
  }
}
```

```
Output: Inside method one
          Caught: java.lang. IllegalAccessException: demo
```

Exceptions Handling: try-catch-finally blocks

- When exceptions are thrown, execution in methods takes an abrupt, nonlinear path that alters the normal flow through the method. It can cause the method to return/stop prematurely.
- The **finally** block is designed to address/handle this behavior.
- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- The **finally** clause is optional, however each **try** statement requires at least one **catch** or a **finally** clause.

Exceptions Handling: try-catch-finally blocks

The following is the general form of an exception–handling block:

```
try{
        //block of codes to monitor exceptions
}
catch(ExceptionType1 exObj) {
        //exception handler for ExceptionType1
}
catch(ExceptionType2 exObj) {
        //exception handler for ExceptionType2
}
finally {
        //block of codes to be executed no matter what!
}
```

Exceptions Handling: try-catch-finally blocks

- Resource leak
  - Caused when resources are not released by a program
- The **`finally`** block
  - Appears after **`catch`** blocks
  - Always executes
  - Used to release resources

Exceptions Handling: try-catch-finally blocks

- Here is an example program that shows two the usage of finally clause.

```
class FinallyDemo {
        static void procA ( ) {

        try {
         System.out.println (“Inside procA”);
        throw new RuntimeException (“demo1”);
        } finally {
          System.out.println (“procA’s finally”);
        }
    }
```

Exceptions Handling: try-catch-finally blocks

```
static void procB () {
        try {
         System.out.println (“Inside procB”);
        } finally {
          System.out.println (“procB’s finally”);
        }
    }
public static void main (String [] args) {
          try {
          procA ();
        } catch (Exception e) {
        System.out.println (“Exception caught”);
        }
          procB();
        }
```

Exceptions Handling: try-catch-finally blocks

Output:
```
Inside procA
procA's finally
Exception caught
Inside procB
procB's finally
```

The ‘Try-with-Resources’ Construct

```
import java.util.Scanner;
public class ClosingAResourceUsingFinally {
public static void main(String[] args) {
Scanner keyboard = new Scanner (System.in); // open a Scanner resource
try {
System.out.println("START TRY");
String[] colours = {"RED","BLUE","GREEN"}; // initialise array
System.out.print("Which colour? (1,2,3): ");
String pos = keyboard.next();
// next line could throw NumberFormatException
int i = Integer.parseInt(pos);
// next line could throw ArrayIndexOutOfBoundsException
System.out.println(colours[i-1]);
System.out.println("END TRY");
}
// include a catch only for ArrayIndexOutOfBoundsException
catch(ArrayIndexOutOfBoundsException e) {
System.out.println("ENTER CATCH ");
System.out.println(e);
}
// this block will always be executed
Finally {
System.out.println("ENTER FINALLY");
keyboard.close(); // Scanner resource closed
System.out.println("Scanner CLOSED");
}
System.out.println("Goodbye");
}  }
```

The ‘Try-with-Resources’ Construct

- **finally** clause can be used to close a resource before exiting a program.

- Once the finally code is executed, any uncaught exception is reported.

- Note that if the code in the finally clause itself throws an exception it is this exception that is thrown from the method rather than the original offending exception.

- The close method of Scanner, for example, would itself throw an IOException if the system was unable to close the Scanner resource, this exception will be reported. This is not ideal as the original exception is probably more appropriate to report.

The ‘Try-with-Resources’ Construct

- One of the more recent developments in Java allows to avoid writing finally clauses to close a resource, but instead adapt the try clause for this purpose.

- This is known as try-with-resources. The try-with-resources clause automatically closes a specified resource (or resources) and suppresses any exceptions that might arise from doing so, leaving any original uncaught exceptions free to be reported.

- The program below re-writes the ClosingAResourceUsingFinally program to make use of this try-with-resources clause.
- When using a try-with-resources clause, the resource declaration is included (in this case the creation of a Scanner object) in round brackets straight after the try keyword

The ‘Try-with-Resources’ Construct

```
import java.util.Scanner;
public class ClosingAResourceUsingTryWithResources {
public static void main(String[] args) {
try (Scanner keyboard = new Scanner (System.in)) // open a Scanner resource here
{
System.out.println("START TRY");
String[] colours = {"RED","BLUE","GREEN"}; // initialise array
System.out.print("Which colour? (1,2,3): ");
String pos = keyboard.next();
// next line could throw NumberFormatException
int i = Integer.parseInt(pos);
// next line could throw ArrayIndexOutOfBoundsException
System.out.println(colours[i-1]);
System.out.println("END TRY");
}
// include a catch only for ArrayIndexOutOfBoundsException
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("ENTER CATCH ");
System.out.println(e);
}
// we have removed the finally clause
System.out.println("Goodbye");
}  }
```

Self study

Read Java in two semester book:

1. Chapter 14.1 – 14.6
2. Chapter 19.1 – 19.5
3. Do the self-test questions in chapter 14 & 19.